NASA SOFTWARE SPECIFICATION AND EVALUATION SYSTEM DESIGN FINAL REPORT PART I

SCIENCE Applications INCORPORATED

(NASA-CR-144270) NASA SOFTWARE SPECIFICATION AND EVALUATION SYSTEM DESIGN, PART 1 Final Report (Science Applications, Inc., Huntsville, Ala.) 74 p HC $4.50
CSCL 09B G3/61
N76-22945
Unclas
26772

SAI-77-555-HU

NASA SOFTWARE SPECIFICATION AND EVALUATION SYSTEM DESIGN FINAL REPORT PART I

SOFTWARE VERIFICATION/VALIDATION TECHNIQUES

CONTRACT NAS8-31379

Prepared under the direction of
Mr. Bobby C. Hodges
Marshall Space Flight Center
National Aeronautics and Space Administration

March 19, 1976

SCIENCE APPLICATIONS, INCORPORATED

2109 W. Clinton Avenue, Suite 800, Huntsville, Alabama 35805
(205) 533-5900

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF FIGURES

LIST OF TABLES

SAI

1. INTRODUCTION

The main body of this summary is divided into ten subsections. This division corresponds closely to the items within the original scope of work (SOW) of the contract. Generally there are two or three consecutive subsections pertaining to each SOW item. The exact correspondence between the subsections and the SOW items is given in Figure 1-1.

The original purpose of this contract was to conduct exploratory analysis within the individual areas mentioned in the scope of work. However, in time the work evolved into the design of a software specification and evaluation system. This system will be denoted as SSES throughout this report. The various techniques and tools which constitute this system are the topics of subsections of section two of this report.

SSES GOALS / SOW ITEMS	EARLY PROGRAM FEASIBILITY AND TESTING	REQMTS & SPECS TRACEABILITY	RELIABLE CODE IMPLEMENTATION	SUFFICIENT TEST CAPABILITY	SUFFICIENT MAINTENANCE CAPABILITY	OTHER
A1. Program Proof of Correctness		(I,2.2,2.3)	(I,2.4)			
A2. Program Flow-path Analysis				(I,2.7,2.8)		
A3. Data Base Verification			(I,2.6)	(I,2.6)		
A4a. Static Analysis			(I,2.5)	(I,2.5)		
A4b. Structural Analysis				(I,2.7,2.8)		
A4c. Dynamic Analysis				(I,2.7,2.8)		
A4d. Testcase Generation				(I,2.9,2.10)		
A4e. Instrumentation				(I,2.7)		
A4f. Production		(I,2.2,2.3)	(I,2.4)			
B1. Early Testing	(I,2.10)					
C1. Evaluation						(I,3), (II, 1)
C2. Specifications						(II,2.,3.,4)

SAI 0483

Figure 1-1. Subsection/SOW/SSES Goal Correspondence

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

2. SSES METHODOLOGY

2.1 GENERAL

Eight months ago, SAI and NASA jointly began a software research effort to develop methods which will reduce the effort expended (usually 50%) in a typical software test and verification effort. The initial emphasis was toward developing techniques that would allow efficient automatic verification of software without a consequent loss in user confidence. Such a verification procedure could be constructed to perform consistently only if stringent demands and checks were put on the software throughout its development. The software development stages generally are understood to be requirements, specifications, code, test, and maintenance. We concluded that an entire software development methodology would have to be developed to ensure that at no stage of the development could major inconsistencies remain undetected, and that from requirements onward the software product would have to be quite formal. The design of our methodology is centered about five goals:

1. Early Program Feasibility and Testing
2. Requirements/Specification Completeness and Traceability
3. Reliable Code Implementation
4. Sufficient Test Capability
5. Sufficient Maintenance Capability

Achieving these ends demanded the development of a formal software requirements methodology, a formal specifications language which could traceably embody requirements, a high level programming language which could be easily and faithfully generated from specifications and could promote a logical error-free code implementation, a language preprocessor to allow compatability of the methodology with existing compilers, and finally, automatic code analysis tools to attain our original objective - that of reducing software test and verification effort. The way in which these techniques would be used in a software development project is depicted in Figure 2-1.

The construction of our unified methodology demanded that each tool and technique take allowance of the other techniques which would be employed prior to and following it. Our original work goal - the allowance of a reduction in software test effort without a corresponding loss in user confidence - meant automatic test tools had to be employed and their functions first had to be defined. After realizing this implication, we concluded that the efficacious performance of these test tools made certain demands on the programs analyzed and on the very programming language employed. Similarly, program language restrictions have placed demands upon the way in which specifications for those programs were to be written; and specification restrictions have an impact upon requirements which would give rise to those specifications. Hence, it was decided that the development of our software methodology had to be carried out in reverse order from that of software development.

Such a methodology - an integrated Software Specification and Evaluation System (SSES), is being developed for NASA/MSFC. Some SSES components are entirely original - like the software specifications language, the data base verifier, and testcase generator (and indeed the whole integrated system itself), while some are improvements upon already existing technology - like the structured programming language and the static analyzer. The Executive Summary does not attempt to treat in depth any one SSES component but only to present the technical highlights and unification of our software methodology. Detailed discussions of the various system components will be the topics of Part II of this report.

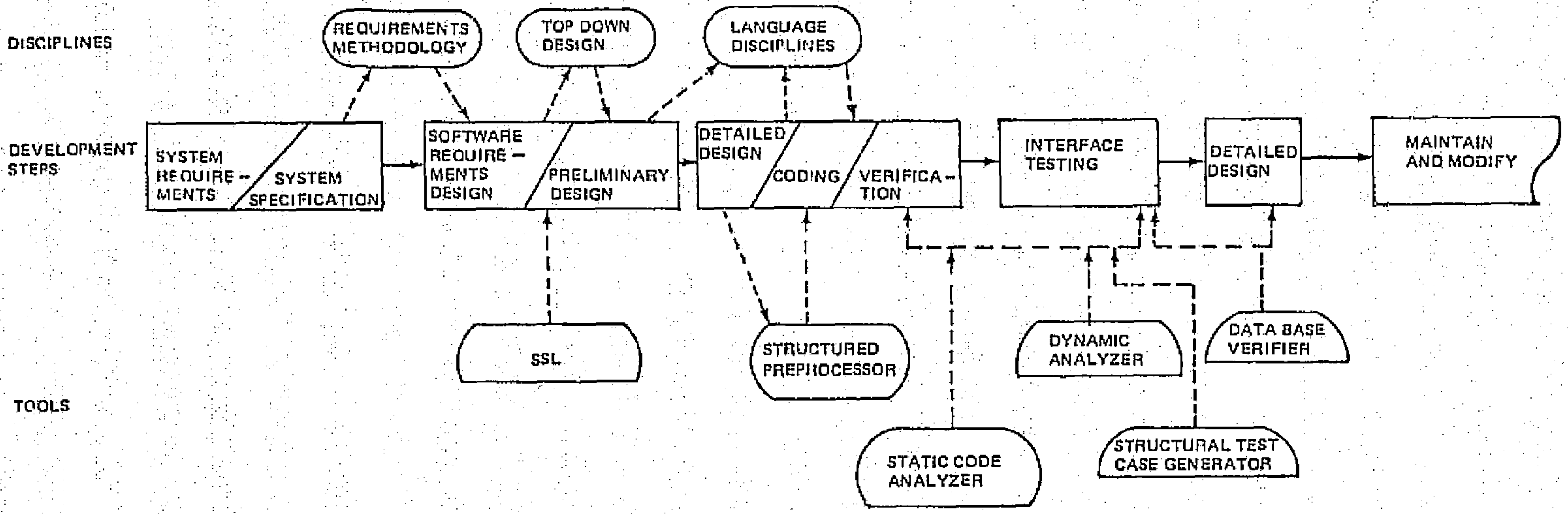


Figure 2-1. Augmented Development Cycle

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

2.2 SOFTWARE REQUIREMENTS METHODOLOGY

The basis for the software requirements methodology was initiated within the September 18th progress report of this contract. Subsections of that report dealing with the software requirements decomposition methodology have been edited for inclusion below.

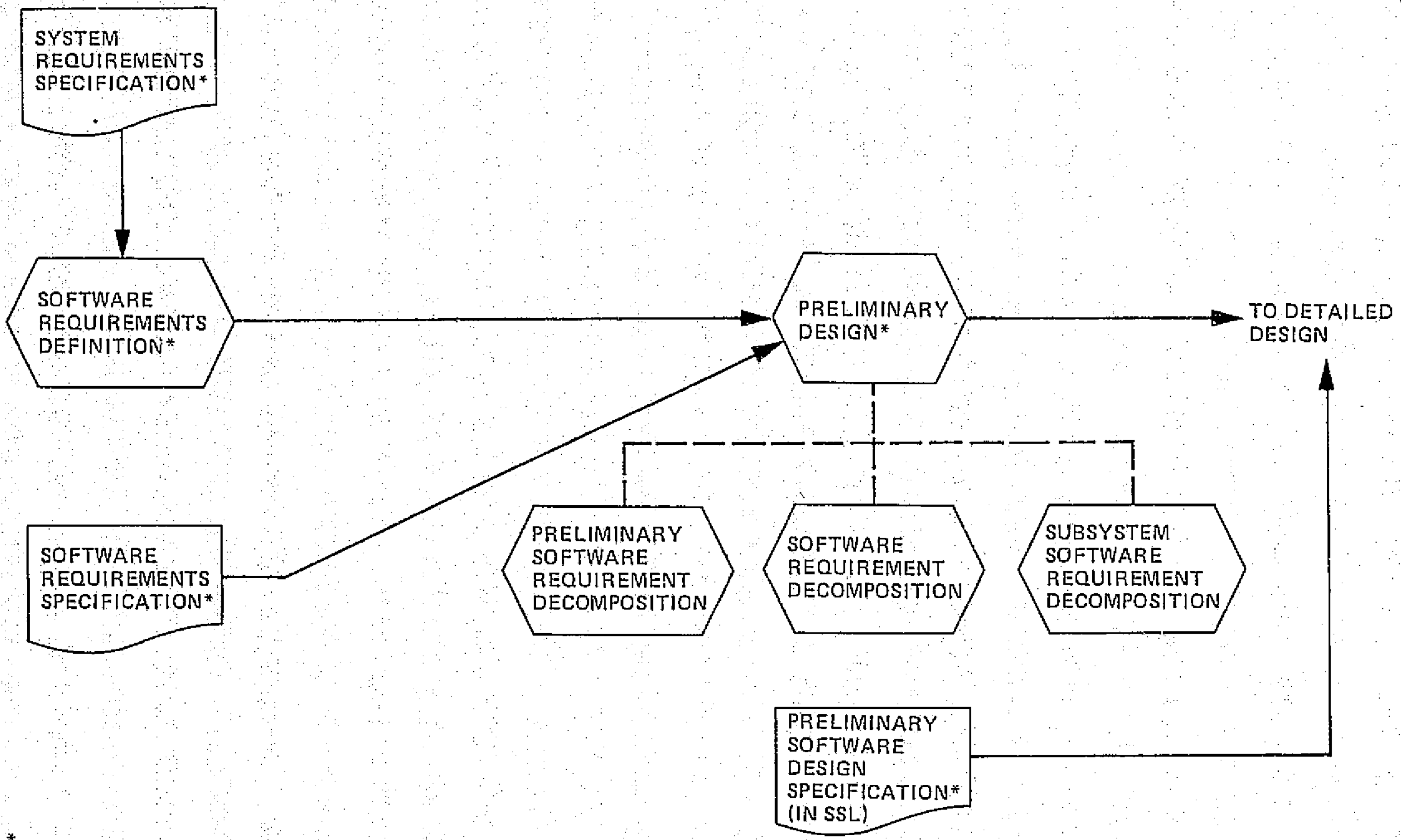
2.2.1 Software Requirements and Traceability

Our efforts toward the semantic definition of a formal software specification language have necessitated an analysis of the manner in which software requirements are stated. In subsection 2.2.1.1, we present our view of the early development stages which is compatible with currect NASA procedures and guidelines. Subsection 2.2.1.2 discusses requirements decomposition in more detail, listing the necessary elements of a decomposition. Subsection 2.2.1.3 presents an example. The names that have been assigned the components are working titles, subject to change.

2.2.1.1 Introduction

Requirements analysis is a continuing effort from problem recognition to problem statement to solution recommendation. It is the final phase, solution recommendation, that is of importance to the process of software module specification. In this report, we are not concerned with how the recommendation was derived, but only how it is stated. In effect, we are stating our assumptions concerning what information is contained within a requirement and recommending a format to make that information explicit. To do this requires that we state our view of the activities and deliverables associated with the early stages of design.

Figure 2-2 illustrates the activities and documents currently depicted in NASA working papers on software development procedures. (Activities are represented by hexagons). We have divided the NASA activity "Preliminary Design" into several subactivities as will be presently explained.


SYSTEM REQUIREMENTS SPECIFICATION*
SOFTWARE REQUIREMENTS DEFINITION*
PRELIMINARY DESIGN*
TO DETAILED DESIGN
SOFTWARE REQUIREMENTS SPECIFICATION*
PRELIMINARY SOFTWARE REQUIREMENT DECOMPOSITION
SOFTWARE REQUIREMENT DECOMPOSITION
SUBSYSTEM SOFTWARE REQUIREMENT DECOMPOSITION
PRELIMINARY SOFTWARE DESIGN SPECIFICATION* (IN SSL)

*FROM NASA WORKING PAPER RECOMMENDATION.

Figure 2-2. Recommended Software Project Time Line

SAI-0507

Principal documents and activities of the early phases of development are:

Systems Requirements Specification - A document describing the functional and environmental characteristics of the problem solution.

Software Requirements Specification - A document which describes the software interfaces with the environment in which it performs as well as key assumptions and constraints.

Software Requirements Definition - An activity whose purpose is to derive the Software Requirements Specification from the Systems Requirements Specification.

Preliminary Software Design Specification - A document describing the data structures and software functions which will comprise the finished software system.

Preliminary Design (PD) - An activity whose purpose is to derive the Preliminary Software Design Specification.

Preliminary Software Requirements Decomposition (PSRD) - A subactivity of PD whose purpose is the declaration of the actions to be performed by a a single software package; emphasis is on what to do and not how (nor how well) to do it.

Software Requirement Decomposition (SRD) - A subactivity of PD whose purpose is the detailed declaration of the actions to be performed by a single software package arranged in the same general form as the PSRD; initial estimates are included that pertain to "when", "how well", and "how" various actions are to be performed.

Subsystem Software Requirement Decomposition (SSRD) - A subactivity of PD whose purpose is a declaration with the same form and substance as the SRD except it pertains to only one subsystem within the software package.

The next subsection contains a detailed summary of the constituents of PSRD, SRD, and SSRD.

2.2.1.2 Contents of the Software Requirement Documents

In this subsection, we state in detail the (quite similar) PSRD, SRD, and SSRD subactivities. In doing so we have included only those facets of a requirement that directly affect the software organization. Noticeably absent are such requirement categories as:

- Manpower and schedules
- Applicable documents
- Acceptance criteria.

We wish to emphasize once more that our purpose is to state the information derived from Software Requirement Decomposition and not the methodology employed.

For the purpose of module specification there are two types of requirement decomposition subactivities. The first is the Software Requirement Decomposition which is an expansion of PSRD. The SRD embraces the recommended solution of the original problem obtained from the customer. The second type is the Subsystem Software Requirement Decomposition which nearly always expands on a subsystem or real time process derived from the SRD or a preceding SSRD. There may be from none to several SSRD activities and they are performed by the analyst at any time up to the conclusion of detailed design. There will be a broader discussion of subsystems within the example of subsection 2.2.1.3. Within a SRD or SSRD there are seven divisions:

Direction - A general statement of the boundaries of the problem.

Transductions - A list of processes to be performed, each of which translates a stimulus into a response.

Input - Data or documents available to the software system from external sources.

Output - Data or documents produced by the software for external purposes.

Constraints - A list of capabilities, design objectives, or resources to be observed.

Preconceptions - A list of specific design alternatives to be observed.

Implications - A binary relation existing between certain transductions.

Of the seven divisions, only the direction and a subset of the transductions are required for the Preliminary Software Requirement Decomposition. (The specific subset of the transductions necessary will be shown in the next subsection). If there are no implications, all transductions are assumed to be independent. Figure 2-3 illustrates the relationship between software and subsystem requirements for a particular system.

2.2.1.3 A Software Requirements Example

Assume that an employer wishes to establish a list of employees based on proximity of residence for the purpose of carpooling. He desires the results compiled in two formats: (1) an alphabetical list of employees and their assigned carpool number, and (2) a list of carpools with individual participants. With the aid of an analyst, he derives the following Preliminary Software Requirement Decomposition:

<u>Direction</u>

Construct carpool lists by individual and by carpool number.

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

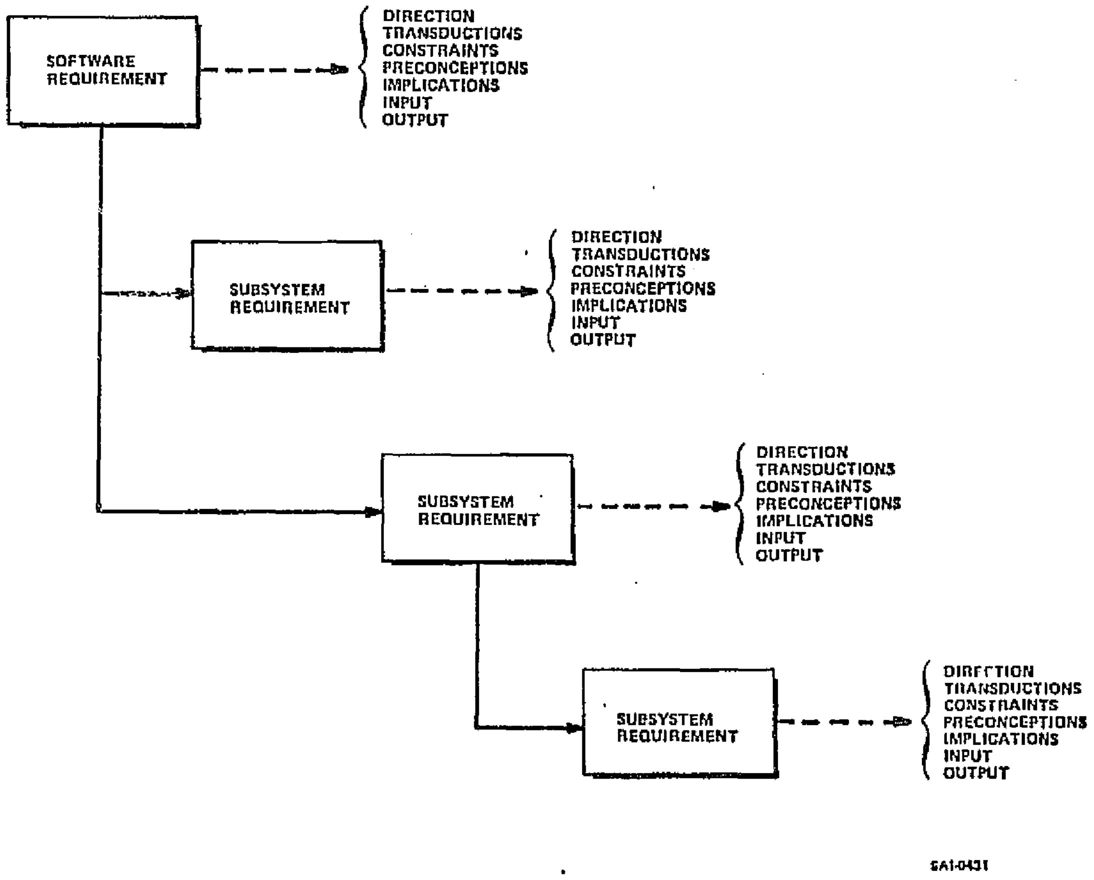


Figure 2-3. Requirements Document Structure

Transductions

Translate:	Transform each employee street address to a uniform coordinate system.
Group:	Cluster employees based on proximity of residence.
Assign:	Assign each individual to exactly one carpool.
Print 1:	List individuals alphabetically with carpool assignment.
Print 2:	List carpools with individuals assigned.

Completeness and unambiguity is necessary in the above statement, but we make no recommendation for attainment other than inspection. The form and content are important for the purposes of software specification. If desired, it may be expanded via footnotes or appended subparagraphs. Particularly important to this problem is a list of available documents such as employee file and street/coordinate tape.

The analyst now assumes primary responsibility and attempts to add, perhaps in consultation with the customer, the following items:

- Constraints that are both problem oriented and computer oriented
- Additional transductions that are either implied by the original transductions or are made necessary by the constraints
- The implication list
- The preconception list
- The input and output lists

The results of this analysis (which complete Software Requirement Decomposition) are stated below:

Transductions (Derived from the previous transductions)

Precord:	Print an employee name and address.
Find:	Find the most eligible individual to add to an existing carpool.
Span:	Compute the minimum tree span of a set of nodes specified via planar coordinates.
Match:	Search the street/coordinate file for the nearest point to a given address.
Reject:	Delete all employees for which street name or street number is not on coordinate file.
Sorte:	Sort carpool file.

Input

Employee file:	Name and address of each employee.
Coordinate file:	Correlation of street addresses with an (x,y)-coordinate system.

Output

List 1:	An alphabetical list of employees with carpool assignments.
List 2:	List of carpools with individuals assigned.

Constraints

Memory:	Use no more than 32K words
Machine:	Use IBM S360/65 for development
Language:	Use ANSI Standard FORTRAN without the arithmetic IF
Size:	No carpool may contain more than five persons
Distance:	The sum of the distances associated with the edges of a minimum spanning tree of residences of members of a single carpool must be less than two kilometers.

Implications

Translate	$\supset$	Match	Print1	$\supset$	Precord
Translate	$\supset$	Reject	Print2	$\supset$	Precord
Assign	$\supset$	Find	Reject	$\supset$	Precord
Assign	$\supset$	Span	Print1	$\supset$	Sorte

Preconceptions

Sort: Use a shell sort to produce the Print 1 listing

Presume that the analyst determines that clustering should be implemented as a major independent subsystem. (He/she may make this decision at any point prior to completing the design.) The desired subsystem will:

(1) Cluster the employee file based on proximity of residence

(2) Write the clusters onto a segmented file, one cluster per segment

(3) Sequentially read the cluster file with end of segment markers.

Commensurate with these goals, the analyst next performs the SSRD. The results are as follows:

Direction

Cluster and order an employee/coordinate file.

Transductions

Initialize:	Find the n farthest apart points in the file to use as initial cluster centroids; n will be an input parameter.
Cluster:	Match each point in the file to the nearest centroid.
Restart:	Compute the centroids of each cluster.
Segment:	Order clusters into segments and save.
Fetch:	Fetch the next element of the current cluster.

Mark:	Test for end of cluster.
Measure:	Find the point nearest a given point from a given set of points.
Swap:	Exchange two employee records

Input

Point file:	A file containing a sequence of (x,y)-coordinates with a unique identifier attached to each

Output

Neighbor:	An entry from the point file
Eos:	An end of segment marker
Eof:	An end of file marker

Constraints

Number:	$n \leq 50$ (the number of clusters).
Halt:	Continue attempts to cluster until the centroids remain unchanged on two consecutive tries.

Implications

$\text{Measure} \subset \text{Initialize}$

$\text{Measure} \subset \text{Cluster}$

$\text{Swap} \subset \text{Segment}$

Preconceptions

None

The analyst now embarks upon the second phase of Preliminary Design, module specification. However, he/she may continue to reduce portions of the system to subsystems. Subsystems are similar to levels of abstraction [1] and have four distinctive characteristics:

(1) The modules within a subsystem do not share any global data (e.g., files or COMMON) with modules not in the subsystem.

(2) The modules located at the subsystem entry points are referenced (called) only by modules not in the subsystem.

(3) No module referenced (called) directly by a module within the subsystem is ever referenced directly by modules not in the subsystem.

(4) All subsystems satisfy one or more of the following criteria:

   (a) Information hiding - The subsystem isolates design decisions likely to change.

   (b) Resource management - The subsystem has exclusive control of particular resources such as a peripheral or data structure.

   (c) Division of labor - The subsystem is logically complete, apart from the original requirements, i.e., it is a reuseable component.

   (d) Real time process - The subsystem operates as an asynchronous activity within a real time application.

2.3 SOFTWARE SPECIFICATION LANGUAGE

Within the course of this contract, a formal language was designed for the purpose of conveying the Preliminary Design specifications of software. In the following subsections, we discuss the purpose, goals, and specific attributes of our Software Specification Language (SSL) as well as provide a simple example of how SSL is used.

2.3.1 Purpose and Goals of SSL

In the software development process, the step between producing software requirements and constructing a detailed specification of the code has been informally supported by charts and diagrams, procedural languages, descriptive systems, and K-diagrams. Therefore, we have developed a software specification language, the function of which is to formally describe the overall software system (or functional) structure, and thereby provide a firm foundation for the aforementioned software development step. Additionally, SSL fulfills another primary goal, the goal of traceable requirements, by incorporating the capability to tag requirements and attach them to specific software objects. In Table 2-1, the goals for our functional specification language are presented along with a brief discussion of each goal.

One of the specified goals in Table 2-1 is "formality." Formality (i.e., rigorous definition) is necessary for automation. Specific attributes of SSL that fulfill this general goal or attribute are:

- A context-free grammar representable in Backus-Naur Form
- Semantics that are defined via set theory

Table 2-1. Functional Specification Language Goals

Specifications should describe the overall software system structure.

Functional specifications should provide a link between the requirements specification and detailed design in terms of a non-procedural description.

Specifications should be formalized.

Formalization permits automatic consistency checking, restricts the designer to the level of detail appropriate for functional specifications, and improves communication between designer(s) and implementer(s) and among implementers.

Specifications should impact reliability.

A formalized specification system is a step toward "designing in" rather than "adding on" reliability and can be accomplished by providing feedback to requirements for design decisions occurring early in the development process. It can assist in ascertaining the correctness of succeeding project development steps rather than relying on external machanisms to remove anomalies.

Specifications should be transparent.

By "transparent" we mean that no reasonable software structure should be rendered impossible to depict due to limitations of the language used. Transparency is necessary so that specification requirements (such as the incorporation of an existing software package) will be possible.

Specifications should be complete and unambiguous.

By completeness, it is meant that all objects created by the designer are subsequently traceable to a set of terminal objects that are provided within the language. Specification ambiguity can arise in structure,

Table 2-1. Functional Specification Language Goals (con't.)

if the object interrelationships can be interpreted in more than one way, or if the basic objects of the language are not well defined.

Specifications should have a uniform level of detail.

The non-procedural nature of a functional specification language will limit the descriptive potential of the language to interconnection relationships.

A specification language should be machine independent.

Machine dependencies are seldom a desirable level of detail. Thus, a specifications language should emphasize conceptual and abstract objects, although recognizable as components of most digital computers.

Specifications should depict interfaces simply and clearly.

In addition to depicting interfaces, the language should be organized in a manner that encourages the designer to construct simple, smooth connections. A specification language should require explicit creation and usage declarations of all objects. It should afford clear and concise rules for assigning attributes to each object and to delineate scope of access.

Specifications should be modifiable.

The degree of ease with which a description can be modified is partly a function of how it is used by the designer. A specifications language could assist by providing mechanisms to isolate and encapsulate critical design decisions. A language should provide methods for a natural partitioning of the problem space and provide means for the designer to create sub-partitions.

A specifications language should be easy to use.

By easy to use, we mean readable by higher levels in the command chain (managers) and easily communicable to implementers.

Table 2-1. Functional Specification Language Goals (con't.)

Achieving this goal would encourage utilization of the methodology, abate clerical errors, assist communication, and provide permanent documentation.

Specifications should reflect error handling requirements.

When project development schedules are underestimated, misjudging the magnitude of error analysis required is generally a contributory factor. A specification language should incorporate explicit declarations of necessary software error checks.

Specifications should reflect fault tolerant capabilities.

The term "fault tolerant" means the ability to cope with errors by the user and in the environment. As in error handling, explicit declarations should accompany I/O accesses.

Specifications should reflect the original requirements.

Generally, the more detailed the design, the more isolated the designer is from the original requirements. A formal specification language could alleviate this problem by permitting requirements to be labeled and attached as attributes to system objects.

Another goal is the enforcement of a uniform level of detail. This is desirable with respect to the top-down programming philosophy and in assuring that equal attention has been given all aspects of the design. Specific SSL attributes that are commensurate with this goal are:

- Utilization of non-procedural language constructs only, to focus attention on static structure rather than algorithm dynamics.
- Adoption of the software module as the elementary unit of definition; a module is one or more compilation units (e.g., subprograms, procedures).

Examining the entire general attribute list provided in Table 2-1 confirms that the specific attribute list of SSL would be necessarily quite extensive and will therefore not be presented in this summary report. A cursory view of SSL is given in the next subsection.

2.3.2 Partitioning of Specifications

Functional specifications for a software system may be divided into three areas; environment, data, and control. SSL provides the capability to specify the minimal set of hardware characteristics that are inherent in the problem definition and that impact software organization (see ACCESS statement in Figure 2-5 of subsection 2.3.3). In the data area, SSL offers the mechanisms to explicitly describe a variety of data structures and to specify when the data is used as input (i.e., in the USES clauses illustrated in Figure 2-5 in subsection 2.3.3) or as output (i.e., in the CREATES or MODIFIES clause shown in Figure 2-5 in subsection 2.3.3). Moreover, SSL is used to describe the module/data interconnection structure and a rationale for the module/data interconnection structure and a rationale for the

structure based on requirements. Regarding the control area, SSL is not designed to depict the control flow within modules. However, intermodule connections can be depicted by use of the EXECUTES statement (in Figure 2-5 of subsection 2.3.3) in which conditional, iterative, or recursive execution of modules is specified. In addition to providing specifications in these areas, SSL ensures that the resulting specifications are sufficiently abstract to prevent selecting a specific machine representation.

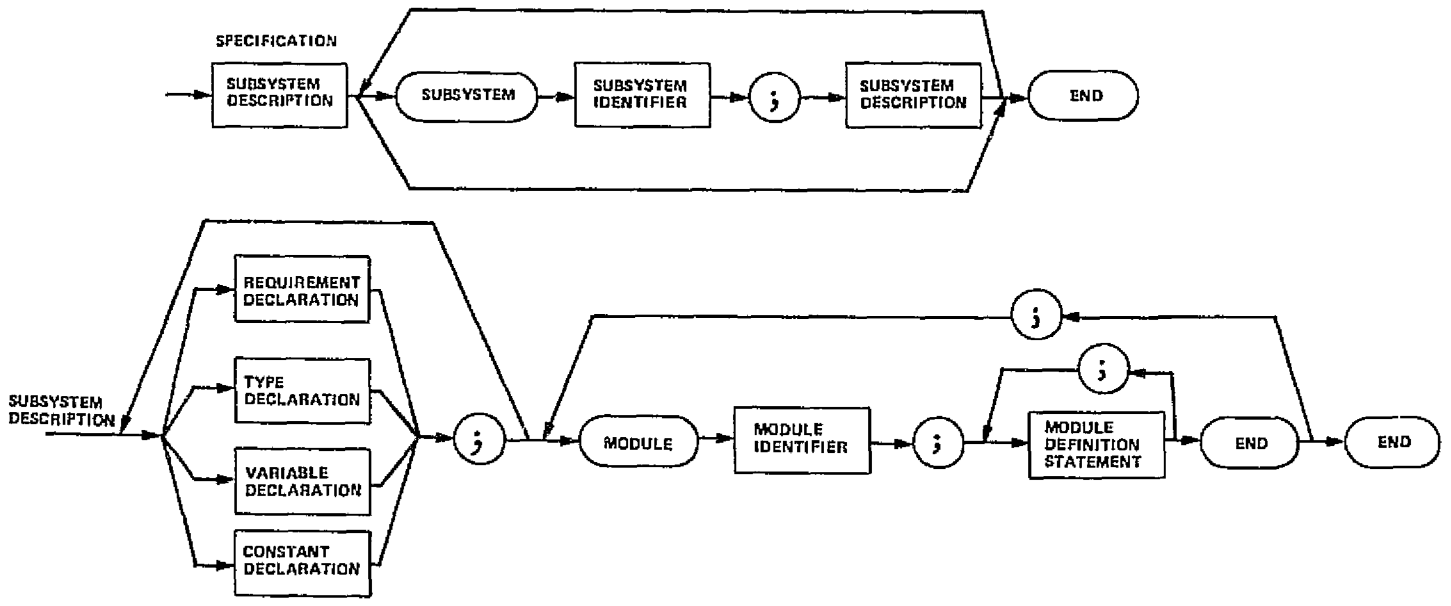
2.3.3 SSL Subsystem and Module Descriptions

SSL allows for partitioning the software into subsystems based on the principle of levels of abstractions. A specification in SSL is represented as a set of subsystems which is shown in Figure 2-4. Each subsystem is defined by a preamble and one or more module descriptions. The preamble describes the local environment for the subsystem and includes: the subsystem name, the requirements associated with it, data types, variables, and constants used within it.

Modules are basic system objects in an SSL system description. In Figure 2-5, we present a portion of the SSL grammar for module descriptions. A correspondence between the module items identified by SSL and the specific statements used to implement these are shown below:

Module Item	SSL Statement
● module name	MODULE, ENTRY
● input data	USES
● output data	CREATES, MODIFIES
● conditions placed on data upon entry to and exit from the module	ASSUMES, SATISFIES,
● dependence on environmental objects and other modules	ACCESSES, EXECUTES
● requirement attributes	FULFILLS

In Figure 2-6, we present a simple example of SSL design for a module.



SAI-0094

Figure 2-4. Syntax Diagram of an SSL Specification

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

```
[MODULE]
[ENTRY ]          < MODULE NAME > ;


        ASSUMES          < ASSERTION LIST > ;

        FULFILLS         < REQUIREMENT LIST > ;

        USES             < DATA OBJECT LIST > ;

        ACCESSES         < ENVIRONMENT OBJECT LIST > ;

        CREATES          < DATA OBJECT LIST > ;
                         [ USING < DATA OBJECT LIST >] ;


        MODIFIES         < DATA OBJECT LIST > ;
                         [ USING < DATA OBJECT LIST >] ;


        EXECUTES         < MODULE NAME LIST > ;

        SATISFIES        < ASSERTION LIST > ;
```

Figure 2-5. SSL Skeletal Definition for Modules

```
MODULE SORT  (N:INTEGER) ;
   /*  MODULE TO SORT ARRAY */
   /*  ARRAY IS INITIALIZED FROM CARD READER  */

   ASSUMES          N  >  0 ;
   FULFILLS         ORDERED _ VALUE;
   ACCESSES         CARD_READER;
   MODIFIES         SARRAY USING N;
   SATISFIES        FORALL  (I:INTEGER)
                    I  >  0  AND  I  ≤  N-1
                    AND
                    SARRAY [I]  ≥ SARRAY  [I+1]

END;
```

Figure 2-6. Description of Module in SSL

2.4 LANGUAGE DESIGN FOR RELIABLE PROGRAMS

The consistent production of reliable computer programs makes stringent demands upon the selection of the programming language employed. For example, to minimize the effort required to carry out a program proof of correctness, the language control structures should be as simple in concept and as few in number as possible. To produce programs that are clearly understood and easily modified, one should construct code in modular units using the top-down philosophy. We suggest the following attributes as being worthwhile goals in the selection of a machine processable language:

- Simple to use
- Easy to understand
- Quickly Machine Processable
- Reliably Machine Diagnosible
- Translatable into Efficient Machine Code

These general language attributes have direct implication upon the structure of the programming language employed. Some of these implications are as follows:

- The language should follow naturally from a top-down approach and should be able to reflect the problem at hand
- The language promotes a sequential implementation
- Control structures should be clear and explicit and should be kept to a minimum
- The language should exhibit the same syntax structure for semantically similar constructs

- The language should allow indentation and a type of modularization that clearly defines the boundary of each module and allows each module to be clearly and completely locally understood
- The language should have meaningful reserved words
- The language should allow the programmer to write often used constructs with a minimum of detail
- The language should offer a non-restrictive placement of comments which facilitates trouble free usage
- Side effect changes of data should be made explicit and restricted to a minimum
- Data types and other information crucial to correct execution should be explicitly specified preferably in several different ways
- The language should have context-free syntax
- The language should allow amenability to automatic code analysis tools
- Machine overhead of often used constructs should be kept to a minimum

Maximal compatibility with FORTRAN - the language most used by NASA - demands that we try to achieve the above language goals through a structured FORTRAN preprocessor. The central structures of the preprocessor are being designed to include the above features. A full description of this preprocessor will be included within a follow-up contract. Along with the preprocessor, a programming methodology is being formulated which pinpoints questionable coding practices, for it is quite possible to construct poor software with the best of languages. Many of these practices are defined in subsection 2.5 (and also 2.5.1) on static analysis.

2.5 STATIC CODE ANALYSIS

A static analyzer for ANSI FORTRAN code accepts source program code as input and evaluates the code in a static manner, i.e., the program being checked is not in execution. A static analyzer can be used to accomplish a variety of functions. After combining our ideas of static analysis with those in the available literature [2] – [6], we identified three areas into which all the functions of a static analyzer can be classified:

- Reliability Enhancement

  The static analyzer could enforce technical coding standards, i.e., the identification and characterization of critical areas and items in the code which are likely candidates for inconsistencies and errors. The NASA tool FACES is directly concerned with this function.

- Verification Determination

  The functional specifications written in SSL (Software Specification Language) can be verified by a static code analyzer, i.e., the consistency of the program code with certain specifications can be checked. This static verification could compare variable and module interconnections of the program with SSL specifications.

- Documentation Assistance

  The documentation of pertinent program information, which will be used during the code testing/debugging and maintenance phases of the software development, can be provided by a static code analyzer.

Pertaining to reliability enhancement, we constructed in Table 2-2, a list of catagories for source code program checking.

Table 2-2. Source Code Program Checks

A.	Syntactical and semantic checks which involve evaluation by element, expression, and statement
B.	Logical structural checks which involve analysis of the program as a single entity and of the entire system of programs as a whole
C.	Machine independence checks
D.	Clarity enhancements (such as requiring nested DO loops to have unique targets)

Performing these checks would provide a comprehensive source code analysis as to style, format, and structure. As illustrated in the following subsection, our efforts in static analysis have been concentrated in this area.

In reference to verification determination, we ascertained that the effort necessary to incorporate the capability of matching source code with SSL specifications was beyond the scope of this contract. However, we recommend this task for a future project since it would represent a significant step toward automated software verification.

We believe that a static code analyzer could expedite the code maintenance process by providing a comprehensive program report consisting of the items in the following table.

Table 2-3. Program Report Items

A. Language element categorization

B. Subprogram cross reference listing

C. Variable usage (i.e., type) inconsistency flags

D. COMMON summary

E. Variable or array initialization summary

F. Special variable role summary
- Adjustable array dimension
- DO loop control variable
- Assigned GO TO variable
- Computed GO TO variable
- Input/output unit designator

G. Input/output reference summary

However, due to higher SSES priorities, implementation of this aspect of static analysis is not currently planned.

2.5.1 Static Code Analyzer Enhancements

During the contract period, documentation for the NASA tool FACES (FORTRAN Automated Code Evaluation System) became available.. Shown in Table 2-4 are the capabilities currently

TABLE 2-4. PRESENTLY KNOWN CAPABILITIES OF FACES

Non-executable Statement Checks

- Subroutine, function and COMMON BLOCK names are not FORTRAN "reserved" words or ANSI standard function names
- All COMMON BLOCKS are checked for alignment, i.e., corresponding elements in COMMON must agree in number, type, dimension, name, and size
- All DATA statements involving COMMON BLOCK variables which are not in BLOCK DATA are detected
- All parameter lists are checked for alignment, i.e., corresponding parameters must agree in number, type, and dimension

Executable Statement Checks

- Potential cyclic calling patterns among routines are flagged
- A DO loop index can not be used outside the loop
- A DO loop variable or parameter should not be redefined within the loop
- Function subroutines should not alter input parameters
- Two-way, three-way, IFs and computed GO TOs should have the next sequential statement as one of the targets
- An uninitialized variable and array element search is performed
- Occurrences of local variables in assignment statements are flagged

featured in FACES. New capabilities deemed desirable and feasible by NASA and SAI are shown in Table 2-5. The detailed specifications for incorporating these new capabilities into FACES are provided in Part II of this final report.

TABLE 2-5 NEW FACES CAPABILITIES

EQUIVALENCE and EXTERNAL statements are flagged.

COMMONs not named are flagged.

ALL COMMON BLOCK arrays must be dimensioned in COMMON BLOCK statements.

DIMENSION statement and variable which contain an adjustable (variable) dimension are flagged.

Constants, hollerith, or arithmetic expression arguments used in subroutine argument lists are flagged.

All occurrences where the same variable exists in multiple positions in an actual parameter list are flagged.

Targets of branches should not be other branches, especially single GO TOs.

TABLE 2-5 NEW FACES CAPABILITIES (Cont.)

Variable which is I/O unit designator is flagged.

Statement labels must appear in increasing order.

Arithmetic IFs are flagged.

Occurrences of error-prone FORTRAN statements such as ASSIGN statement, assigned GO TO, and PAUSE are flagged.

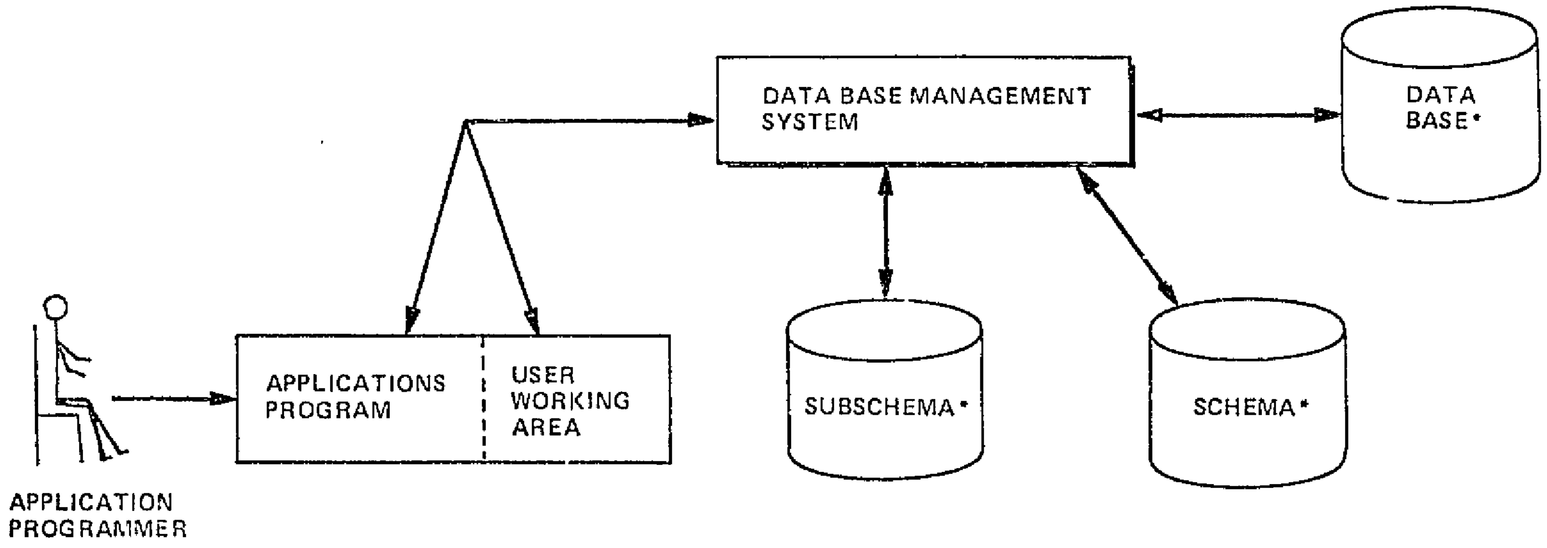
The appearance of the same COMMON variable in more than one DATA statement is flagged.

2.6 DATA BASE VERIFICATION

The problem of verifying the structure and contents of a stored data base (i.e., the part of a data base which resides on permanent storage) is difficult. However, the problem becomes more complex when it is coupled with the task of ensuring the continued integrity of the stored data base throughout accessing and updating operations. Since the late 1960's the CODASYL (Conference of Data Systems Language) and other organizations have been engaged in the formalization of their approach to these and other problems concerning data bases [7]. The CODASYL has directed its efforts toward developing language standards for describing extensions to existing high level languages (such as FORTRAN) which will allow access and operation on the data base components as well as describe the part of a data base which resides on permanent storage.

As background for discussing our approach to data base verification, we present CODASYL's view of a data base management system. A data base management system is a system which manages and maintains data in a nonredundant structure for the purpose of being processed by one or more applications. In the environment depicted in Figure 2-7, an applications programmer writes a program in a high order programming language such as FORTRAN or COBOL which has been extended to incorporate Data Manipulation Language (DML) commands. The DML statements are data base access mechanisms, i.e., they provide application program interfaces to the data base during execution. (Note: CODASYL's usage of the term "data base" is the same as our definition of stored data base.)

A schema DDL (Data Description Language) completely defines the data base; it includes the names and descriptions of



*BUILT AND MAINTAINED BY THE DATA BASE ADMINISTRATOR

Figure 2-7. Architecture of a CODASYL Data Base Management System

all the areas, set types, record types and associated data items and data aggregates as they exist in the data base [8] and are known to the data management programs. However, an applications program must be concerned with the description of only that part of a data base which is useful and meaningful to it. This description is called a subschema. A subschema is a subset of a schema which allows the applications program to view only those portions of the data base declared necessary for that particular program. Therefore, the remainder of the data base is insulated from the execution of an applications program or a subsystem of programs. The characteristics of the data items (and the arrangement of items within records) described by the subschema may be different from the characteristics of those data base items defined by the schema. Since a program depends only on the subschema for data base information, changes may be made to the schema of the data base and the data base may be appropriately adjusted without affecting the programs using the data. Correspondingly, a subschema may be modified to provide compatibility with a specific programming language, and the schema will not be affected.

The actual mapping or conversion of subschema descriptions to schema definitions is performed by the data base management system (DBMS). (The subschema contains the mapping definition which specifies the correspondence between the subschema and schema.) Thus a degree of data independence is established by employing the schema and subschema mechanisms. At the same time, flexibility in the choice of programming languages is supported since that part of a data base known to a program can be described according to any particular programming language conventions.

For each application program, there is defined a user working area (UWA) which contains locations for all data delivered to the program from the data base and vice versa. The program refers to these locations via names in the subschema.

In fact, the UWA is set up by the DBMS according to the subschema which is evoked by each applications program [9].

In the process of creating and maintaining a data base management system, the need for human involvement becomes apparent. The human activities are performed by the data base administrator (DBA). According to CODASYL, the DBA is responsible for:

- Writing the schema and subschema
- Modifying the schema and subschema to reflect changing user needs
- Designing, assembling, and loading the data base
- Monitoring the use and performance of the data base and reorganizing the data base for greater performance efficiency if required.
- Assigning data to physical devices
- Assigning privacy locks and issuing privacy keys to users for specific portions of the data base
- Recovering the data base after system malfunctions.

Since the DBA is responsible for data management as indicated by the above functions, the application programmer is relieved of this responsibility and can concentrate on other aspects of programming.

This discussion of the interworkings of the DBMS components provides a basis for a series of data base verification subsystems which when integrated would form a data base verification system. The data base verifier that we designed can be considered as one of these subsystems. Our data base verification subsystem concentrates on the FORTRAN applications program which, according to CODASYL, must be written in ANSI FORTRAN that is extended to incorporate Data Manipulation Language (DML) commands.

As input, our data base verifier accepts CODASYL FORTRAN Data Manipulation Language. The specifications for this language are still being refined and will not be finalized until the end of this calendar year. However, we obtained a CODASYL FORTRAN Data Base Facility Journal of Development which was printed on November 25, 1975. Though this document is only a working paper for the FORTRAN Data Base Manipulation Language Committee, we used it as the foundation of our design, since the basic problems to be analyzed and solved will remain constant though the syntax may be altered by the committee during the refinement stages.

A brief summary of the functions of our data base verifier are the following:

- Accepts FORTRAN DML source code as input
- Statically analyzes the program and constructs tables which describe the stored data base that the program accesses and manipulates
- Prints a summary of all the information collected about the components and the structure of the stored data base.

The user must then establish the consistency and validity of the stored data base within the framework of the program descriptions by cross referencing these tables. A future enhancement to our data base verifier includes the automatic consistency checking of these data base descriptions as set forth by the applications program. Part II of this report contains functional specifications (i.e., SSL descriptions) of the data base verifier subsystem.

2.7 GRAPH ANALYSIS AND INSTRUMENTATION

The key to most program analysis systems is the selection of a model which yields the correct program characteristics to base analyses on. The standard approach that is used in systems built for software testing is to use a program-graph model. We will describe the general process of formulating a program-graph, and present the pertinent manipulations which support a dynamic analysis system. In the first section, the procedure for forming the program-graph is described. The next section contains a description of DD-PATHS in a program-graph.

2.7.1 The Program Graph

A program-graph is formed from a program unit (main procedure or subroutine) by mapping selected program statements into nodes and corresponding edges. To illustrate the program graph construction, we present in Figure 2-8 an exhaustive decomposition of the elements in the ANSI FORTRAN. This type of language construction could be carried out for any language. The set of abbreviations on the right hand side serve a two-fold purpose: (1) they identify the particular statement types that must be addressed; (2) they provide a convenient short form to refer to each of these statement types. Note that compound IF statements are completely decomposed in this table so that all constituent parts can be identified; for example, IF5 refers to a statement of the form

```
IF (logical expression) IF (arithmetic expression) n,m,p.
```

The set of FORTRAN elements directly referenced in Figure 2-8 are the only elements which are mapped into the program-graph.

There are nine mapping formulae. In each of these cases the statement is mapped directly into a node; decisions are made only concerning the formation of edges from that node.

• Executable	
• Assignment	
• Arithmetic	AS1
• Logical	AS2
• GOTO	AS3
• Control	
• GOTO	
• Unconditional	GO1
• Assigned	GO2
• Computed	GO3
• IF	
• Arithmetic	IF1
• Logical	--
• Simple	
• With GOTO	IF2
• With RETURN	IF3
• Other	IF4
• Compound	
• With Arithmetic IF	IF5
• With Computed GOTO	IF6
• With Assigned GOTO	IF7
• CALL	CA1
• RETURN or END	RE1
• CONTINUE	CO1
• STOP or PAUSE	ST1
• DO	
• With Unique Terminal	DO1
• With Shared Terminal	DO2
• I/O	IO1
• Non-Executable (Headings)	
• Main Program Implicit Begin	HE1
• SUBROUTINE	HE2
• FUNCTION	HE3

Figure 2-8. ANSI FORTRAN Statement Table

1. Direct Sequential, Single: A single edge is formed between the given node and the next sequential node.

2. Direct Sequential, Double: Two edges are formed as in #1.

3. Termination: A single edge is formed to the designed terminal node.

4. Direct Defined, Single: An edge is formed between the given node and another node specified as its target.

5. Direct Defined, Multiple: Several edges are formed as in #4.

6. Mix 1-4: Two edges are formed; one as in #1, the other as in #4.

7. Mix 1-5: Several edges are formed; one as in #1, the others as in #5.

8. Mix 1-3: Two edges are formed; one as in #1, the other to a node designated as the terminal (as in #3).

9. Loop: Three edges are formed; one as in #1, the second from a designated target node back to the given node, and the third from a designated target node to another designated target node.

In Figure 2- 9, the elements in the statement table are matched with their corresponding mapping formula. Note that the mapping formula simply follow the potential flow of control from each statement. Formula #9 is used for the DO statement; the third edge connects the DO target with the statement which is executed once the DO is satisfied.

The FORTRAN DO statement, following ANSI documentation. is expected to behave so that the loop index is set to the first parameter, the loop is executed. and then the index incremented and checked against the terminal parameter. In this fashion, it appears that control is centered in the DO target statement rather than the DO itself; hence, the edges are formed as described in #9.

Formula.	Elements
1. Direct Sequential, Single	AS1-3, CA1, CO1, ST1, IO1, HE1-3 [IF NONE OF THE ABOVE ARE THE DO TARGET -- IN WHICH CASE #9 APPLIES]
2. Direct Sequential, Double	IF4
3. Termination	RE1
4. Direct Defined, Single	GO1
5. Direct Defined, Multiple	GO2, GO3, IF1
6. Mix 1-4	IF2
7. Mix 1-5	IF5, IF6, IF7
8. Mix 1-3	IF3
9. Loop	DO1-2

Figure 2-9. Mapping Formula Against Statement Types

The program-graph is formed in a two stage process. The first stage involves the lexical scan of the program and the identification of statement type and auxiliary pointers. These pointers reference the node(s) to which the given node is to be connected when such nodes are not the next sequential node number. Consider as an example the statement

```
IF (A.GT.B)  IF  (C)  10, 20, 30
```

which in this scheme would be identified as IF5; the pointers which must be determined are three in number; pointer to the node which corresponds to label 10, the same for label 20, and for label 30.

The DO loop poses special problems since the target statement takes on the semblance (from a program-graph viewpoint) of a control statement. The usual procedure for a DO is then to identify the target statement via a pointer. Some confusion arises when several DO statements share a common target (DO2); hence, it is recommended that in this case, an artificial target is added to allow only DO1's to occur in the program.

The program-graph for a sample program, Figure 2-10, is shown in Figure 2-11.

2.7.2 DD Paths

The program-graph representation accurately reflects the control flow within a program. It is also the case that there is a reduced form for the program-graph which also captures that control flow. The DD Path program-graph is formed from the program-graph by collapsing linear segments of the structure. A linear segment is a series of nodes which have a single edge in and a single edge out.

```
        Text                                     Type      Node

        SUBROUTINE SAMPLE                        HE2       1
        SET = SET + 1.                           AS1       2
        ASSIGN 10 TO LABEL                       AS3       3
        GOTO LABEL                               GO1       4
   20   CALL OUT                                 CA1       5
        IF (EXP) 20,30,40                        IF1       6
   30   SET = SET + 2.                           AS1       7
        IF (EXP1) GOTO (10,20,30,40), I          IF5       8
        GOTO 20                                  GO1       9
   10   ISET = ISET + 3                          AS1       10
        CALL OUT                                 CA1       11
        DO 50 J = 1,9                            DO2       12
        SET = SET + 1.                           AS1       13
        DO 50 K = 5,6                            DO2       14
        CALL OUT                                 CA1       15
   50   CONTINUE                                 CO1       16
        GOTO 20                                  GO1       17
   40   RETURN                                   RE1       18
```

Figure 2-10. Sample Program

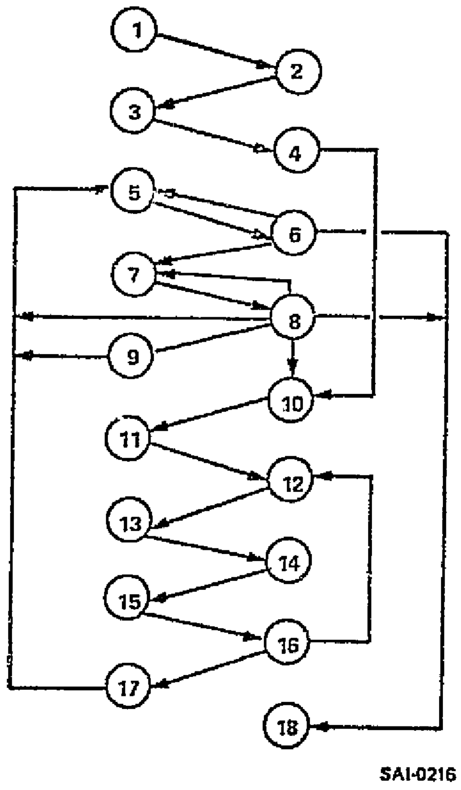


Figure 2-11. Program-Graph for Sample

The flow of control within a program is induced by the program branches; likewise the flow within a program-graph is induced by the branch nodes (nodes with more than a single emanating edge). A DD Path is a path in the program graph which begins and ends on a decision node (D node). The D nodes are the branch nodes, the entry node, and the terminal node; in the sample program-graph (Figure 2-11) the D nodes are:

1, 6, 8, 16, 18

corresponding to statements

HE2, IF1, IF5, CO1 (DO2), RE1

The table in Figure 2-12 is the collection of DD Paths for the sample program-graph. The DD Path graph is shown in Figure 2-13.

Each DD Path begins with a decision node; however, the actual path is a collection of edges and each edge corresponds to an outcome from a node. A DD Path is, then, representative of a set of outcomes from each D node; there is a condition associated with the selection of each DD Path. The contents of a DD path are simple to derive, and since all but the first node have only one successor node, it is easy to describe each DD path by its constituent nodes without loss of information.

A point to be discussed in the next subsection is that the DD Paths are identified by numbers and the numbers are assigned in a manner which is unique (it can be relied upon to give the same numbering each time). This numbering algorithm is the following:

1. Order the DD paths by their initial node
2. Within each group of DD paths with the same initial node order the elements by their correspondence to the D node decision outcome. The ordering by outcome is:

REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

DD PATHS

No.	From:To	Contents	Condition
1	1:16	1-2-3-4-10-11-12-13-14-14-16	entry
2	6:6	6-5-6	EXP .LT. 0
3	6:8	6-7-8	EXP .EQ. 0
4	6:18	6-18	EXP .GT. 0
5	8:16	8-10-11-12-13-14-15-16	EXP1 .EQ. TRUE .AND. I .EQ. 1
6	8:6	8-5-6	EXP1 .EQ. TRUE .AND. I. EQ. 2
7	8:8	8-7-8	EXP1 .EQ. TRUE .AND. I. EQ. 3
8	8:18	8-18	EXP1 .EQ. TRUE .AND. I. EQ. 4
9	8:6	8-9-5-6	EXP1 .EQ. FALSE
10	16:16	16-12-13-14-15-16	DO @ 12 Loop
11	16:16	16-14-15-16	DO @ 14 Loop
12	16:6	16-17-5-6	DO nest completion

Figure 2-12. DD Paths for Sample

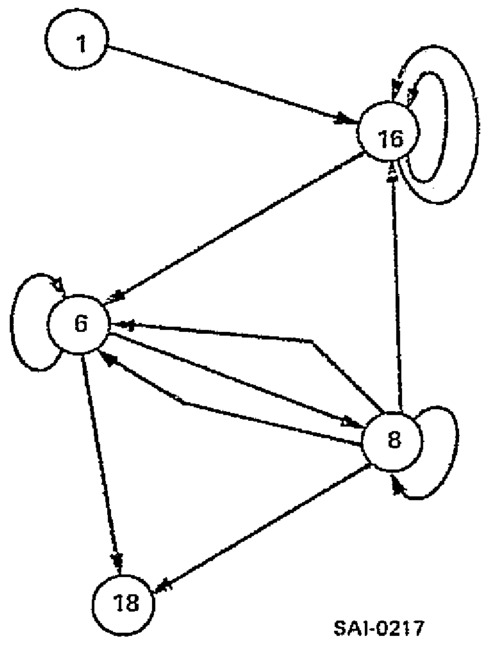


Figure 2-13. DD Path Graph for Sample

2.1 True before False

2.2 If there are several trues (or falses) order them by the lexical order of their target definitions in the original program statement.

For example, the statement

```
IF (EXP1) GOTO (10, 20, 30, 40), I
```

in a program would obviously have 5 DD paths stemming from its assigned node; these paths would be ordered by

a. EXP1 .EQ. TRUE .AND. I .EQ. 1

b. EXP1 .EQ. TRUE .AND. I .EQ. 2

c. EXP1 .EQ. TRUE .AND. I .EQ. 3

d. EXP1 .EQ. TRUE .AND. I .EQ. 4

e. EXP1 .EQ. FALSE

Aside from the above description of the ordering procedure as external source views it, there is a simpler internal description. Since statements are scanned sequentially in order, the edge list is formed in order by beginning node number. The edge list for a single node follows from the lexical scan so that in, say, the above example a, b, c, d are in the natural order. Since the edge list is already ordered, the DD Paths are created in the desired order by simply using the edge list and assigning DD Path numbers in a sequential fashion starting from the first D node in the edge list.

To clarify the above "simpler" internal description, consider the edge list for the sample program. The first several entries of their list would be

1 to 2

2 to 3

3 to 4

4 to 10

5 to 6

6 to 5
6 to 7
6 to 18
7 to 8
etc.

The D nodes, as presented earlier, are 1, 6, 3, 16, 18. The DD Paths would be, then,

1 to 2 to . . . n1
6 to 5 to . . . n2
6 to 7 to . . . n3
6 to 18
etc.

The above sequence of DD Paths obeys the external order standpoint, although it is created from purely straightforward devices.

2.7.3 Probe Numbers and DD Paths

One probe is placed for each DD Path in the program-graph.

The probe numbers are the DD Path numbers. See the following subsection for a complete discussion of probe placement algorithms.

2.7.4 Instrumentation Templates

A software probe is a CALL to an auditing subroutine; the subroutine, in turn, records the identity of the actual probe which evoked it.

Probes are placed in the software so that they can intercept the passage of the program control point. Since paths are induced in the program by the branch statements, it is necessary to position probes only at those branch points. (The exceptions to this rule are the entry and exit points from the program which are advantageous to probe.)

The placement of probes is carried out by a macro-expansion technique for each branch point in the program. As each branch point is determined the branch is replaced (expanded) into a composite program segment which incorporates a probe without loss of the logical capability of that branch. For example, the statement

```
      IF (X .EQ. Y)  A = B+C
```

would be expanded into

```
      IF (X .EQ. Y)  GOTO 99
      CALL PROBE (*)

      GOTO 98
   99 CALL PROBE (*)
      A = B+C
   98 CONTINUE
```

2.7.4.1 ANSI FORTRAN Branch Statements

ANSI FORTRAN branch statements can be considered in two classes: simple and compound. The simple statements are

```
DO
IF  (-THEN)
IF-ARITHMETIC
COMPUTED-GOTO
ASSIGNED-GOTO
```

The compound statements involve the combination of an IF (-THEN) with another branch statement; the most straightforward type to handle is

```
IF-COMPUTED-GOTO
IF-ASSIGNED-GOTO
IF-IF-ARITHMETIC
```

For convenience,we will treat branch statements in the groups described above.

The macros for the given FORTRAN statement types involve the use of auxiliary statement labels. It is assumed that a facility will be maintained which will assign these labels so no program conflicts arise. For the examples shown, these auxiliary labels will be given numbers decending sequentially from 99999.

The exact nature of the arguments which are passed in the probe envocation is not described in this paper; hence the probe CALL will be noted simply by the key word "PROBE."

A. DO

```
      DO  [label]  [index] = [n], [m], [o]

      ....

  [label] [statement]
                    *****
      DO  [label]  [index] = [n], [m], [o]
      PROBE

      ....
  [label] [statement]
      PROBE
```

B. IF (-THEN)

```
      IF  [expression] [non-branch statement or
            simple GOTO]
                    *****
```

```
         IF  [expression]  GOTO 99999
         PROBE
         GOTO 99998
99999    PROBE
         [non-branch statement or simple GOTO]
99998    CONTINUE
```

C. IF-ARITHMETIC

```
         IF [expression]  [label-a], [label-b], [label-c]
                      *****
         IF  [expression]  99999,99998,99997
99999    PROBE
         GOTO  [label-a]
99998    PROBE
         GOTO  [label-b]
99997    PROBE
         GOTO  [label-c]
```

D. COMPUTED GOTO

```
         GOTO  ( [a] , [b] , ... , [z] ),  [index]
                      *****
         [temp]  =  [index]
         PROBE-SPECIAL
         GOTO  ( [a] , [b] , ... , [z] ),  [temp]
```

Note: Since the value of [index] must be an integer and since probes are assigned integer identification numbers, then it is straightforward to create the situation where the probe number can be computed using [index] . The PROBE-SPECIAL is a probe evocation which takes into account the value of [index].

E. ASSIGNED GOTO

```
        GOTO  [index] , ( [a], [b], ...,[z])
                        *****
        [temp] = [index]
        PROBE-SPECIAL
        GOTO  [temp],  ([a], [b], ... ,[z])
```

F. IF-COMPUTED GOTO

```
        IF [expression]  GOTO  ([a], ... , [z]), [index]
                        *****
        IF [expression]  GOTO  99999
        PROBE
        GOTO  99998
99999   [temp] = [index]
        PROBE-SPECIAL
        GOTO  ([a], ... , [z]), [temp]
99998   CONTINUE
```

G. IF-ASSIGNED GOTO

See B and F for obvious solution.

H. IF-IF-ARITHMETIC

See B, C, and F for obvious solution.

2.8 DYNAMIC ANALYSIS

2.8.1 Introduction

Implementation of an automatic code analysis tool can be carried out using the theory developed in subsection 2.7. Such an analysis tool would monitor code execution and would be capable of performing the following functions:

- Indicate unexercised code segments
- Indicate execution statistics for exercised segments within each module
- Indicate execution statistics for whole modules
- Monitor variable principal values and the point within the code these values were attained

These functions would be supplied through probe information supplied by the user. A tenative description of how the user might interface with a dynamic analyzer is described below.

2.8.2 User Interface

The dynamic analysis tool would have two main operational parts. The first part performs syntactic analysis for instrumentation purposes; the second processes and interprets the instrumentation (run-time) data. Due to this natural organization, it makes sense to partition the user interface along the same lines.

The user interface facilities are:

- Commands
  - --- Probe Placement (instrumentation)
  - --- Variable Value monitoring
  - --- Reporting Options
- Reports
  - --- Archival Listing
    - - DDP Identification

- DDP Conditions
- Variables Codes for Reference

--- Coverage Reports

- Effectiveness of Module Testing
- DDP Coverage Per Module
- Variable Value Information

User commands are processed during the first phase (syntactic analyses) and the appropriate capabilities added to the target program. THE USER WILL BE GIVEN THE ADDITIONAL FEATURE OF SOURCE LANGUAGE SELECTION -- ANSI FORTRAN OR STRUCTURED FORTRAN.

According to user commands the archival listing will incorporate ALL syntactic information so that reports can reference DDPs by number, and variables by a symbolic name (or code).

The advantage of this approach is there is no need to save any syntactic program information to carry out a complete coverage analysis. The user can refer to the archival listing for all collaborative information, i.e., DDP elements, DDP conditions, actual variable names, etc.

2.8.3 Commands

PROBE module_name

This command causes the indicated program module to be instrumented for coverage purposes. Module_name equal to "MAIN" causes the main program to be instrumented (FORTRAN only); "ALL" causes all procedures, subprograms, and functions to be probed.

MONITOR variable_name (type) [IN module_name]

This command will monitor the principal values of the indicated variable at all DDP control points in the specified module. MAIN is assumed if no module name is given. The "type" field must contain the type of the variable in the module (INTEGER, REAL, LOGICAL, etc.). The specified module must have been instrumented (explicitly or implicitly) by a PROBE command.

<u>SELECT (ANSIFORT or STRFORT)</u>

Indicates whether an ANSI FORTRAN or structured FORTRAN is being analyzed.

<u>NOLIST module_name</u>

Suppresses the archival listing for a particular module.

<u>NOSUMMARY module_name</u>

Suppresses the test effectiveness summary for a particular module.

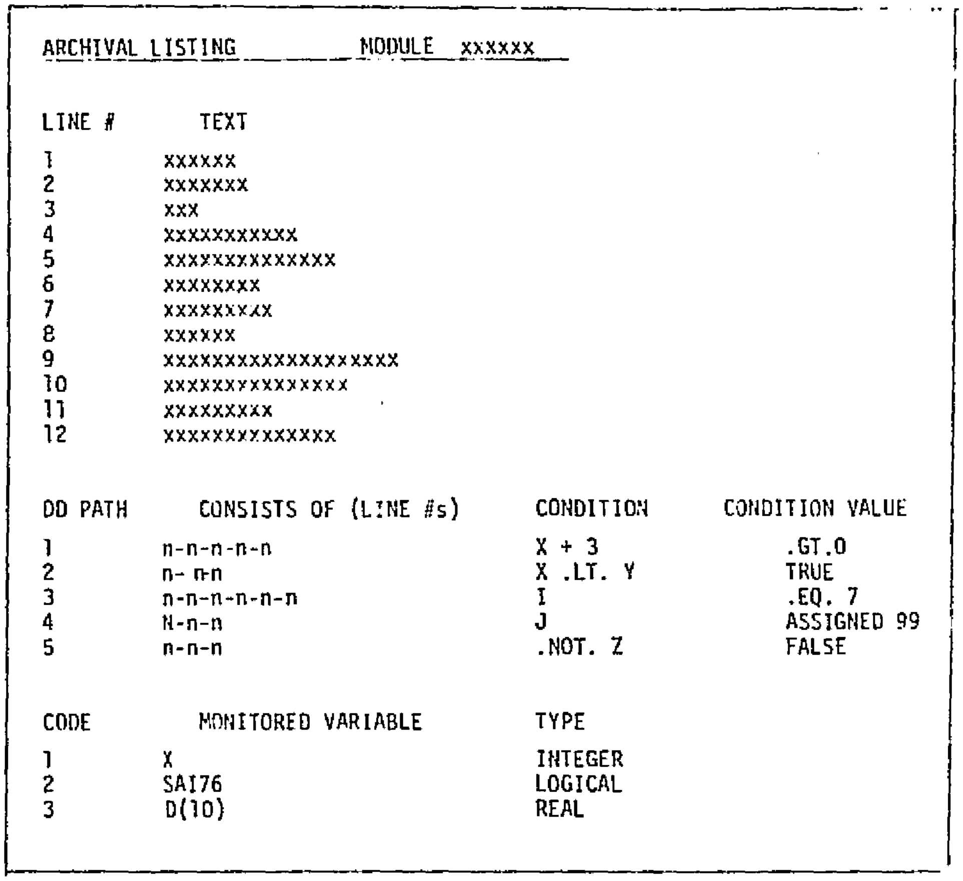
<u>REPORT module_name [VARIABLES]</u>

Causes the printing of a detailed DDP coverage report for the indicated module, with or without variable monitoring information (see below).

2.8.4 <u>Reports</u>

Figures 2-14, 2-15, and 2-16 show sample formats of the various reports.

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR



```
ARCHIVAL LISTING          MODULE  xxxxxx

LINE #        TEXT
1          xxxxxx
2          xxxxxxx
3          xxx
4          xxxxxxxxxxx
5          xxxxxxxxxxxxxx
6          xxxxxxxx
7          xxxxxxxxx
8          xxxxxx
9          xxxxxxxxxxxxxxxxxxx
10         xxxxxxxxxxxxxxx
11         xxxxxxxxx
12         xxxxxxxxxxxxxx

DD PATH       CONSISTS OF (LINE #s)         CONDITION       CONDITION VALUE
1          n-n-n-n-n                        X + 3                .GT.0
2          n- n-n                           X .LT. Y             TRUE
3          n-n-n-n-n-n                      I                    .EQ. 7
4          N-n-n                            J                    ASSIGNED 99
5          n-n-n                            .NOT. Z              FALSE

CODE          MONITORED VARIABLE            TYPE
1          X                                INTEGER
2          SAI76                            LOGICAL
3          D(10)                            REAL
```

Figure 2-14. Sample Archival Listing

```
MODULE TESTING EFFECTIVENESS SUMMARY

MODULE    TIMES INVOKED      # DD PATHS*     #EXECUTED*    % COVERAGE

 xxx           1                 7              5             71.5
xxxx           2                 5              3             60.0
xxxxx          7                 3              3            100.0

             TOTAL              15             11             73.5

*(AT LEAST ONCE)
```

Figure 2-15. Summary Report Sample

```
DETAILED TEST REPORT               MODULE xxxxx

MODULE  xxxxx WAS INVOKED 1 TIME

DD PATHS     TOTAL COVERAGE              PROFILE
                                         (Percent Executed)      100%

1                    10                  **
2                    20                  ****
3                    10                  **
4                    10                  **
5                    50                  **********
6                     0
7                     0

TOTALS

7                   100                  PERCENTAGE DD PATHS 71.5

VARIABLE MONITORING

CODE      INITAL     FINAL      MINIMUM    MIN DDP    MAXIMUM    MAX DDP

1         0          0          0          1          5          4
2         17.6       -3.05      -20.6      1          17.6       1
3         TRUE       FALSE        -          -          -          -
```

Figure 2-16. Detailed Module Test Report

2.9 AUTOMATIC TEST CASE GENERATION

2.9.1 Testcase Generation Functions

The theory presented in section 2.7 helps form the basis for the design of an automatic test case generator. Functions performed by this automatic tool would include:

- Facility to indicate input variables of the program
- Facility to estimate total number of executable paths
- Facility to determine a minimal collection of paths for execution of all code
- Facility to determine paths which execute selected code segments
- Facility to generate data to exercise specific code segments

The first three capabilities can be obtained directly through the graph formation and manipulation techniques described in in section 2.7. The last two capabilities are more difficult and, in a certain sense, impossible. We shall describe the sense in which an automatic test case generator can assist the user in generating test data.

2.9.2 Language Considerations

The test case generator would work optimally on programs written in a well designed language. The basis of such a language, in the form of a structured FORTRAN preprocessor, has been formed in section 2.4. The control structures of that language were chosen to insure clarity and minimality, while retaining as much of the versatility and compactness of the original FORTRAN as possible. For example, the ordinary FORTRAN DO loop is replaced by

```
FOR     i = i1, i2, [i3]

                code

END  FOR
```

As opposed to FORTRAN, this loop executes if and only if i_2 is greater than i_1. In a similar way, if one uses a well designed language, all segments of code are associated with a particular condition either on control indices and/or program variables. (Such conditions are, in fact, recorded in the archival listing of the dynamic analyzer.) The testcase generator uses all the conditions associated with a particular path to attempt to generate input data to exercise that path. If one choose a particular module within the code, the testcase generator attempts to determine feasible paths which reach from a program input position to that module; and next it attempts to generate data to exercise that path. We need to emphasize here that building a program to guarantee solving a system of simultaneous equations and inequalities is theoretically impossible However, a program can be built which in most cases can succeed in generating at least one set of correct data.

2.9.3 Theoretical Foundations

A new approach to program testing, called symbolic execution [10] could help form the basis for a testcase generator. It describes, in terms of original input variables, the actions of the program's successive processing steps. Assume input variables to be $x_1 \ldots x_n$ and that after k processing steps, the first branch statement is reached, which in terms of $x_1 \ldots x_n$ is of the form

```
IF    F  ($x_1$ - $x_n$)  .GT. 0
         THEN   GO   TO   20
         ELSE   GO   TO   30
END  IF
```

The testcase generator tries one acceptable input value of the vector $x_1 \ldots x_n$, say $a_1 \ldots a_n$. If $F(a_1 \ldots a_n) > 0$ we would attempt to alter $a_1 \ldots a_n$ so that $F<0$, to exercise the other side of the branch. To do this we would compute the negative gradient of F, $-dF(x_1 - x_n)$ and, to decrease the value of F, alter $a_1 - a_n$ by a vector of a predefined length in the directon of $-dF(x_1 - x_n)$. The testcase generator again computes F at the new value and essentially by the steepest descent method, attempts to find a value of $a_1 \ldots a_n$ which would make F negative. Thus, using these simple techniques, we could construct an automatic testcase generator to successively generate data to exercise all program path much like a mouse finding its way through a maze; but the implementation of such a testcase generator has yet to be carried out.

2.10 SOFTWARE REQUIREMENTS TESTCASE GENERATOR

The conception of a computer program - at the requirements stage - should be an optimal time for the establishment of most benchmark testcases by which the program can be fairly assessed. Through two SSES methods already presented we feel the <u>automatic</u> generation of test cases from software requirements is a realistic possibility.

Recall that one of the principal features of the Software Specification Language (cf. section 2.3) is the fact that it tags all software modules with the particular requirements which that module was constructed to fulfill; thus it physically maps requirements into the software. Combining this feature of SSL with the way in which automatic structural testcase generation can be accomplished, we see that automatic requirements testcase generation is a distinct possibility, as is illustrated in Figure 2-17. In selecting a requirement, one, via SSL, also selects all software modules which fulfill that requirement. Then, through the use of the automatic structural testcase generator, one obtains testcases which exercise those modules. Though the implementation of the scheme has yet to be carried out, its feasibility seems to be clear; we feel its value is obvious.

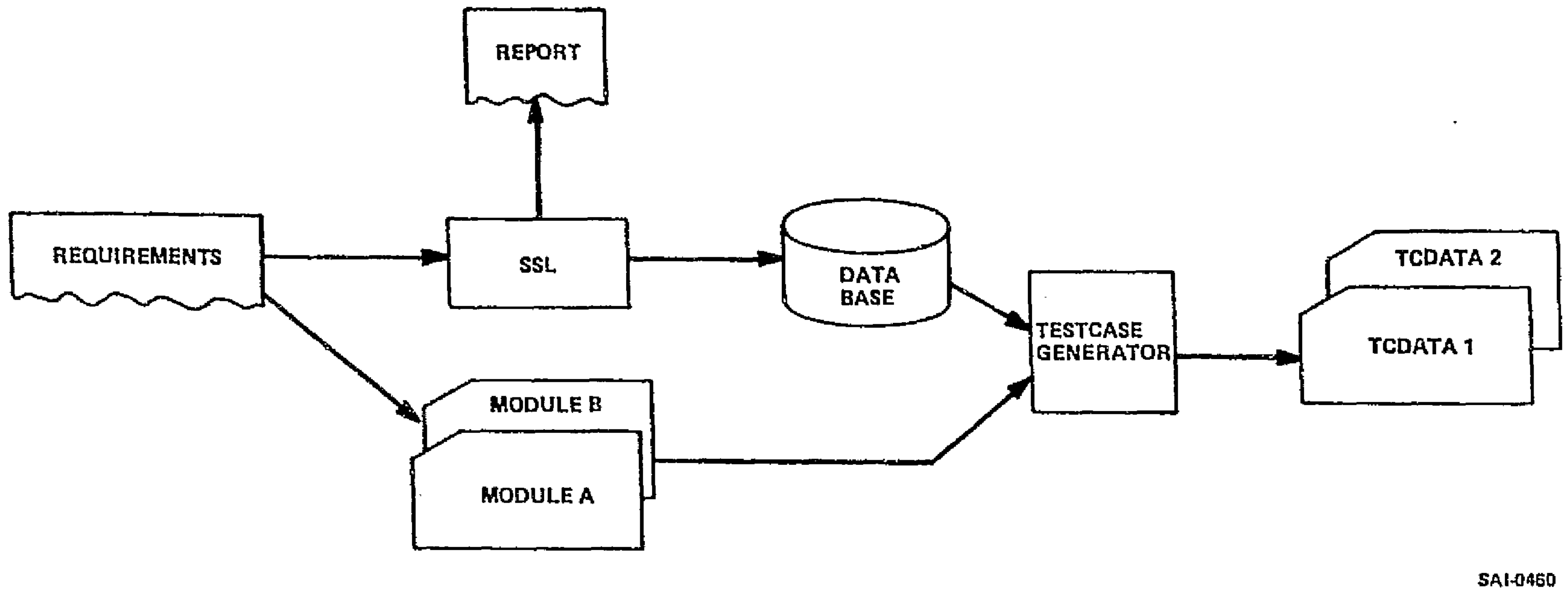


Figure 2-17. Software Requirements Testcase Generation

3. CONCLUSIONS

The aim of the work done under this contract was to make exploratory studies in a variety of research fields which could potentially make valuable contributions to reliability of software. By distilling and summarizing our efforts, we obtain the following advocations about building computer programs:

- The software requirements stage (cf. Figure 2-1) should result in a structured, formal document which leads naturally into the software specification stage. It should be produced by an experienced analyst working in conjunction with the user. Origination of key software testcases should be an integral part of this stage.
- Software functional design specifications should be carried out through a formal language which is capable of reflecting fidelity of design with software requirements.
- Program code should be implemented using a structured programming language in which control structures are operationally apparent and as few in number as tolerable. Hence, a

structured preprocessor should be employed for code implementation if a structured compiler language isn't available.

- A programming language which promotes standardization of methods for accessing and operating on stored data bases such as the CODASYL Data Manipulation Language should be adopted and employed for purposes of data base verification.
- Software testing should be automated to establish user confidence while minimizing cost. Both static and dynamic testing are required. A static analyzer should enforce programming standards, while a dynamic analyzer should check the reliability of the code during execution. Structural and requirements testcase generators would greatly enhance the utility of the analyzers. A structural testcase generator produces data to test as many branches of the code as possible and should be employed for determination of software reliability. A requirements testcase generator produces data to determine the consistency of the code with the software requirements.
- Maintenance documentation needs to be an integral part of software. Documentation guidelines need to be established.

4. REFERENCES

1. Liskov, B. H., "A Design Methodology for Reliable Software," Proceedings of the AFIPS 1972 FJCC, Vol. 41 (1972), pp. 191-199.

2. Lyon, Gordon, "Static Language Analysis", October 1973, Contract Report Number NBS TN-797 for National Bureau of Standards, Department of Commerce.

3. Hsia, Pei, "Survey and Recommendations on Verification Methodologies to Improve Software Quality", March 1975, Contract Report for Computer Sciences Corporation.

4. Kernighan, Brian and Plaugher, P. J., The Elements of Programming Style, McGraw-Hill Book Company, 1974.

5. Ramamoorthy, C. V. and Ho, S. F., "Testing Large Software with Automated Software Evaluation Systems", Proceedings International Conference on Reliable Software, April 1975.

6. Ledgard, Henry F., Programming Proverbs, Hayden Book Company, Inc., 1975.

7. Palmer, Ian, Data Base Systems: A Practical Reference, Q. E. D. Information Sciences, June 1975, pp. 3-5.

8. CODASYL Data Description Language," Journal of Development, June 1973, National Bureau of Standards Handbook 113, page 2.6.

9. Date, C. J., An Introduction to Database Systems, Addison-Wesley Publishing Company, 1975, page 227.

10. King, J. C., A New Approach to Program Testing, Proc. Int. Conf. on Reliable Software, April 1975.